如何快速地找出数据流的中位数?

——2021/8/27 Leetcode.295数据流的中位数

- **前言:**

①你需要的储备知识：堆排序的思想

②用到的语言：C/C++

- **题目:**

> 中位数是有序列表中间的数。如果列表长度是偶数，中位数则是中间两个数的平均值。
>
> 例如，
>
> `[2,3,4]` 的中位数是 `3`
>
> `[2,3]` 的中位数是 `(2 + 3) / 2 = 2.5`
>
> 设计一个支持以下两种操作的数据结构：
>
> `void addNum(int num)` - 从数据流中添加一个整数到数据结构中。
> `double findMedian()` - 返回目前所有元素的中位数。

- **示例:**

```
addNum(1)
addNum(2)
findMedian() -> 1.5
addNum(3)
findMedian() -> 2
```

- **思路:**

拿到这道题，我猛的一看，那简单，用链表做，每加入一个数据，就遍历到相应的位置，插入，记录已有的数据数量。当查询中位数时，根据数量的一般遍历链表，再根据奇偶性做出不同的取中位数方式。果不其然，超时了，不愧是困难题，笑嘻了🤣

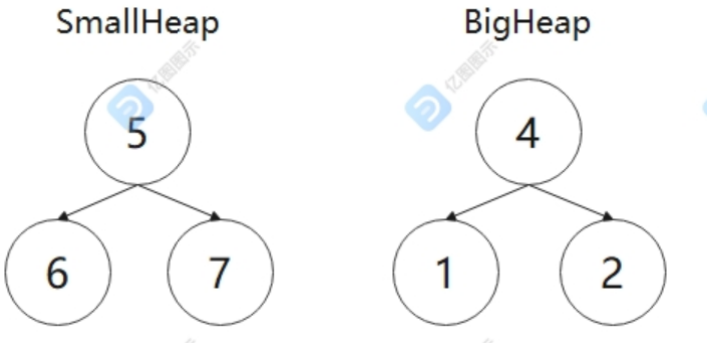
反转了，泪落了下来😭😭😭这都是啥啊。难道还有更好的办法吗?

还真有。试想一下，如果我们每次录入数据时，都尽量让已有的数据分成两部分（奇数情况肯定有一部分比另一部分多一的），并且一部分偏小，一部分偏大，然后两部分都是排好序的，那查询中位数时不就可以快速地通过拿出两部分数据的最小值和最大值求平均即可吗。当然这是偶数时的做法，如果是奇数那更好办了，取出长度最长的一部分，再取出它的最大值/最小值。

将数据分成两部分也是有讲究的，自然不可能1、2、1、2地将数据平均分配到两个部分中，例如[4,1,7,2,5,6]分成[4,7,5]和[1,2,6]，再排个序，便成了[4,5,7]和[1,2,6]，本来平均数是4.5，现在成了5，这是不对的。

因为其实两部分有严格的大小关系（即，一部分的所有数都严格大于另一部分的所有数，反之也成立），所以我们用到了小顶堆和大顶堆来表示这两部分。他们也可以叫小根堆和大根堆，反正一个意思。

话说回来，什么叫小顶堆和大顶堆？字面意思来讲，对于小顶堆，它的顶是整个堆中的最小值；**严格地讲，它是一个完全二叉树**（样子上确实很像一个谷堆🤭），**且每个非叶节点都小于它的孩子**，从下往上堆自然出现顶部最小的现象。大顶堆同理。如图，我们希望构造出如此两个堆。当然，同深度的节点之间的大小不进行比较，因为没必要，毕竟我们只是要顶部数据。



如何构造它们又成了一个问题。**一个想法是，第一个数据放在SmallHeap里，对于后面的每一个数据num，如果num大于等于SmallHeap的顶数据，则将其放入SmallHeap，否则放入BigHeap。为了平均分配，每一次加入新数据后，加入的Heap的数量大于另一边，则将该Heap的顶数据加入另一个Heap中。**由于有奇偶性差异，我们规定奇数时，在SmallHeap中取中位数，则判断是否转移顶部的条件便是：

①SmallHeap->BigHeap：SmallSize > BigSize + 1；

②BigHeap->SmallHeap：BigSize > SmallSize；

- **代码：**

理清楚思路又如何，代码不会打啊呜呜呜😭😭😭

反转了，C++简单的一批🤭，这里用到了**优先队列**。因为顶部出数据，故考虑队列；又由于每个部分的数据都有大小关系，所以加入数据时要根据优先情况插入。

```
class MedianFinder {
public:
    priority_queue<int,vector<int>,less<int>> bigHeap;
    priority_queue<int,vector<int>,greater<int>> smallHeap;

    MedianFinder() { }

    void addNum(int num)
    {
        if(smallHeap.empty() || num >= smallHeap.top())
        {
            smallHeap.push(num);
            if(smallHeap.size() > bigHeap.size()+1)
            {
                bigHeap.push(smallHeap.top());
                smallHeap.pop();
            }
        }
        else
        {
            bigHeap.push(num);
            if(bigHeap.size() > smallHeap.size())
            {
                smallHeap.push(bigHeap.top());
                bigHeap.pop();
            }
```

```
        }
    }

    double findMedian()
    {
        if(smallHeap.size() == bigHeap.size())
        {
            return (double)(smallHeap.top()+bigHeap.top()) / 2;
        }
        else return smallHeap.top();
    }
};
```

说是队列，其实优先队列的内部实现是用二叉树思想的，但是C++加了这层抽象，我们看不到，不管它底层，用就完事了😷

那我缺的C这块谁给我补啊😭😭😭

C模拟这个过程十分困难，对于别的语言真不算困难题，对于C来说是噩梦。接下来的代码，其实算是C++优先队列的底层操作。所以阅读下面的代码要有耐心。

```
#define MAX 100000
typedef struct {
    int num;
    int *p;
} MedianFinder;

MedianFinder *strObj;
/** initialize your data structure here. */

int bigHeap[MAX] = {0};
int smallHeap[MAX] = {0};
int bigSize;
int smallSize;

MedianFinder* medianFinderCreate() {
    strObj = malloc(sizeof(MedianFinder));
    strObj->num = 0;
    strObj->p = bigHeap;
    bigSize = 0;
    smallSize = 0;

    return strObj;
}

/*flag: 1 大根堆 -1 小根堆*/
void UpdataHeap(int *heap, int size, int flag)
{
    int fatherIdx = size - 1;
    int childIdx;
    int originVal = heap[fatherIdx];

    while (fatherIdx >= 0) {
        childIdx = ((fatherIdx + 1) >> 1) - 1;
        if (childIdx < 0 || originVal * flag < heap[childIdx] * flag) { //需要判断是否存在子节点。。
            break;
```

```
        }
        heap[fatherIdx] = heap[childIdx];
        fatherIdx = childIdx;
    }
    heap[fatherIdx] = originVal;

    return;
}

void BigHeap(int num)
{
    bigHeap[bigSize++] = num;

    UpdataHeap(bigHeap, bigSize, 1);

    return;
}

void SmallHeap(int num)
{
    smallHeap[smallSize++] = num;

    UpdataHeap(smallHeap, smallSize, -1);

    return;
}

/*flag: 1 大根堆 -1 小根堆*/
void AdjastHead(int index, int heapSize, int *heap, int flag)
{
    int childIdx;
    int middle;
    int max;
    int originVal = heap[index];
    //保存节点原始值，这里不要用index，heap中的内容后续会改变！！！

    middle = heapSize >> 1;

    while (index < middle) {
        childIdx = 2 * index + 1; //左子节点
        max = heap[childIdx];
        if ((childIdx + 1 < heapSize) && (flag * max < flag * heap[childIdx +
1])) {
            //需要判断是否存在右子节点。。
            childIdx = childIdx + 1;
            max = heap[childIdx];  //选择左、右子节点大（小）值
        }

        if (flag * originVal > flag * max) {
            //满足大（小）根堆条件
            break;
        }

        heap[index] = heap[childIdx];
        index = childIdx;
    }
    heap[index] = originVal;
```

```
    return;
}

void DeleteBigHeapHead()
{
    bigHeap[0] = bigHeap[--bigSize];
    //移除head后，将末元素移到head
    AdjastHead(0, bigSize, bigHeap, 1);

    return;
}

void DeleteSmallHeapHead()
{
    smallHeap[0] = smallHeap[--smallSize];
    //移除head后，将末元素移到head
    AdjastHead(0, smallSize, smallHeap, -1);

    return;
}

void medianFinderAddNum(MedianFinder* obj, int num) {
    /*1、每次增加元素如果大于小根堆最小元素，则加入小根堆；2、如果小根堆个数比大根堆个数多余1
个，则将小根堆顶点移到大根堆*/
    if (smallSize == 0 || num >= smallHeap[0]) {
        SmallHeap(num);
        if (smallSize > bigSize + 1) {
            BigHeap(smallHeap[0]);
            DeleteSmallHeapHead();
        }
    } else {
        BigHeap(num);
        if (bigSize > smallSize) {
            SmallHeap(bigHeap[0]);
            DeleteBigHeapHead();
        }
    }

    obj->num = bigSize + smallSize;

    return;
}

double medianFinderFindMedian(MedianFinder* obj) {
    if (smallSize == bigSize) {
        return ((double)(bigHeap[0] + smallHeap[0])) / 2;
    } else {
        return smallHeap[0];
    }
}

void medianFinderFree(MedianFinder* obj) {
    free(strObj);
    memset(bigHeap, 0, MAX * sizeof(int));
}
```

该代码引用自https://leetcode-cn.com/problems/find-median-from-data-stream/solution/cshi-xian-liang-gen-dui-by-huan-ke-yi-3/，这老哥很厉害，思路清晰😚。代码质量较高，故引用再做详解😁。

- **对于C版本代码的详解，涉及底层实现，建议先阅读过堆排序再来观看：**

（如果你阅读过堆排序的原理，那应该知道整个流程是在数组上模拟完全二叉树的。要熟记父点和子点间的映射关系）

首先，我们需要理解每个函数的用途。

①**UpdataHeap**：我们每加入一个数字到堆里面，都需要维护这个堆的结构——每一个非叶节点都要严格的大于/小于其的子节点。该函数从刚入堆的数字出发，查找它的父点，如果父点和它不满足堆的结构关系，则交换；成为父点，在往上找，如果还不满足，则交换……一直持续这样的历程，直到满足大小关系，break。

②**BigHeap/SmallHeap**：往相应的堆里加入数据，加入后UpdataHeap，维护堆的结构。

③**AdjastHeap**：我也不知道这个老哥的Adjast什么意思🤣，但是这个函数作用很清晰：当将一个堆的堆顶移入另一个堆时，重新维护这个堆，把堆顶的元素按照结构关系往下沉，至于为什么堆顶的元素一定能下沉，看下文。

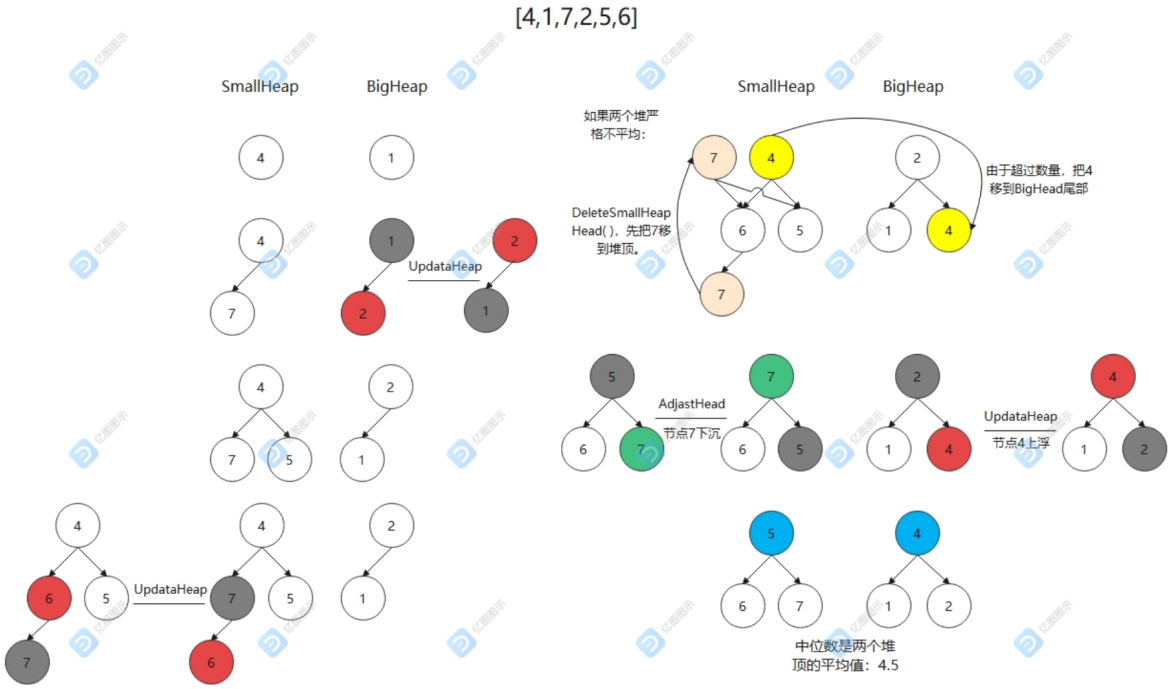
④**DeleteBigHeapHead/DeleteSmallHeapHead**：先把末尾的数字移到第一个位置代替原来的数字（这样就模拟堆顶pop了），然后AdjastHeap维护堆结构。由于将末尾移到前面，原来好不容易沉到底的数字突然到前面，那必是可以下沉了。

⑤后面的Add和Find函数即是我们前面思路的实现。

反正都是一维数组，那为什么维护时不直接排序，而要用到这种二叉树的形式来维护呢？因为一方面兄弟节点的大小关系不重要，谁前谁后不打紧；另一方面，就算是快排也要O(nLogn)的时间复杂度，必超时，还不如我最开始的朴素思想呢😊

总而言之，BigHeap/SmallHeap相当于C++的优先队列的push，
DeleteBigHeapHead/DeleteSmallHeapHead相当于pop。

然后，下面给出了一个数组[4,1,7,2,5,6]做例子，模拟代码过程，图解代码，促进理解：



由于空间问题，我不好放太长的数组，上面的例子上浮下沉每次都是替换一个，但是我们要知道，如果数据很长，它是一直上浮下沉到合适的位置才停下的。

最后，回头再读一边代码，是不是理解了捏😊？~~不是我手，这都不能理解直接入土子吧~~

- **后记：**

第一次用这种软件写笔记，太潮了属于是🤺🤺🤺，写的不好见谅，不清晰的地方和不对的地方请留下你宝贵的指导🙇，我会补充改进的😊。还有大家画这种图都是用什么软件啊，有没有不用米的，我想恰一个😭😭😭😭。